다중프로세서 시스템 환경에서 병렬 루프 스케줄링 알고리즘

이영규[†] · 박두순[††]

요 약

병렬 스케줄링의 목적은 다중프로세서 시스템 환경에서 병렬성을 가진 응용프로그램에 대해 최소의 동기화 오버헤드와 부하균등(load balance)을 달성하도록 스케줄링을 수행하는데 있다. 프로세서들이 병렬 반복(iteration)을 실행하기 위해서는 메모리로부터 반복들에 대한 chunk를 계산하고 할당받게 된다. 이때, 전역 메모리의 상호 배타적인 빈번한 접근으로 많은 스케줄링 오버헤드 및 병목현상이 발생된다. 또한, 프로세서에게 할당된 chunk내 병렬 반복들의 분포가 서로 상이한 경우, 각 chunk의 실행시간이 서로 달라 부하불균등의 원인이 되어 결과적으로 전체 스케줄링 성능에 나쁜 영향을 준다. 따라서, 최소의 스케줄링 오버헤드와 부하균등을 달성하기 위해 본 논문에서는 기존의 방법들에서 문제점들을 도출하고, 자료의 국부성과 프로세서 동족성(affinity)을 고려한 병렬 루프 스케줄링 알고리즘을 제안한다.

A Parallel Loop Scheduling Algorithm on Multiprocessor System Environments

Young-kyu Lee[†] and Doo-Soon Park[††]

ABSTRACT

The purpose of a parallel schedulling under a multiprocessor environment is to carry out the scheduling with the minimum synchronization overhead, and to perform load balance for a parallel application program. The processors calculate the chunk of iteration and are allocated to carry out the parallel iteration. At this time, it frequently accesses mutually exclusive global memory so that there are a lot of scheduling overhead and bottleneck imposed. And also, when the distribution of the parallel iteration in the allocated chunk to the processor is different, the different execution time of each chunk causes the load imbalance and badly affects the capability of the all scheduling. In the paper, we investigate the problems on the conventional algorithms in order to achieve the minimum scheduling overhead and load balance. we then present a new parallel loop scheduling algorithm, considering the locality of the data and processor affinity.

1. 서 론

다중프로세서 시스템(multiprocessor system)환경에서 병렬 스케줄링(parallel scheduling)은 프로세서들을 독립적으로 충분히 활용하여 주어진 응용 프로그램을 얼마나 효과적으로 실행하느냐의 문제이며, 효과적인 실행을 위해 응용 프로그램이 최대의 병렬성을 가져야한다. 응용 프로그램에서 병렬성의 가장 큰 근원은 루프가 될 수 있으며, 루프에 대한 스케줄링 방법들은 수 년동안 계속 연구되어왔다 [2,3,7,8,10,12]. 루프에서 모든 반복들이 처리 순서에 무관하게 동시에 독립적으로 실행될 때, 이를 병렬 루프라 하고, 병렬 루프의 모든 반복들에 대한 분포(distribution : μ, σ)가 균일하여 실행 시간이 비슷한 경우에는 N개의 서로 독립된 반복들을 컴파일 시간에 정적으로 프로세서에게 균등하게 할당하는 것

본 연구는 정보통신진흥원의 '99년 대학 기초연구 지원비에 의한 것임.
† 정회원, 극동정보대학 전산정보처리과 조교수
†† 정회원, 순천향대학교 정보기술공학부 교수

이 효과적일 수 있다. 이런 할당 정책을 정적 스케쥴링이라 한다. 그러나 루프의 body 부분에 조건문이 포함되어 있거나 중첩된 루프 인덱스들의 실행시간이 서로 불규칙적인 경우에는 반복들의 상이한 실행시간 때문에 부하불균등(load imbalance)이 발생되어 정적 스케쥴링은 효과적인 스케쥴링을 수행할 수 없다. 따라서, 불규칙적인 반복들을 효과적으로 실행하기 위해서는 수행 시간에 반복들이 프로세서에게 할당되어야 한다. 이런 할당 정책을 동적 스케쥴링이라 하며, 약간의 스케쥴링 오버헤드가 수반된다. 규칙적인 반복들과 불규칙적인 반복들에 대한 스케쥴링을 효과적으로 수행하기 위해서는 동적 스케쥴링과 정적 스케쥴링이 모두 고려되는 스케쥴링 정책이 필요하며, 이는 병렬 타스크의 최소 종료 시간(minimum finish time)을 달성하기 위한 필수적인 요건이다.

낮은 스케쥴링 오버헤드와 부하균등을 달성하는 최적의 스케쥴링을 위해 고려되어야할 몇가지 요소들을 살펴보면 다음과 같다. 첫째, 각 반복들의 분포를 고려해야 한다. 그러나 각 반복들의 분포가 실행시간에 조차 알려지지 않기 때문에 기존 대부분 스케쥴링 알고리즘에서는 각 반복들을 서로 임의의 독립된 변수(random independence variable)로 취급하여 처리하고 있다[2,7,8,12]. 둘째, 데이터 국부성(data locality)을 고려해야 한다. 이는 서로 인접한 반복들이 가장 근사한 분포를 가진다는 확률적 의미를 반영하는 것이다[9]. 셋째, 각 타스크를 프로세서에게 할당할 때, 소요되는 전송시간이 고려되어야 한다. 이는 네트워크의 구성 형태와 관련이 있으며, 분산스시템에서 반영될 요소이다. 넷째, 각 프로세서들의 성능이 고려되어야 하며, 이종 시스템(heterogeneous system)에서 특히, 반영될 요소라 할 수 있다.

본 논문에서는 프로그램 병렬성의 근원이 되는 루프에 관심이 있으며, 특히, 병렬 반복들의 실행 시간이 매우 불규칙적인 루프(irregular loop)에 초점을 둔다. 불규칙적인 루프는 크기나 타스크의 분포가 입력 데이터에 의존하는 병렬 수행을 내포한다. 이런 프로그램들을 효과적으로 수행하기 위해 본 논문에서는 컴파일 시간에 정적으로 반복들에 대한 자료 국부성을 고려하여 제어키(control-key)에 의해 실행시간 분포가 비슷한 여러 개의 그룹으로 분해한다. 또한, 분해된 그룹들은 프로세서 동족성을 고려하여 각 프로세서의 지역 메모리에 정적으로 할당하고 독립적으로 수행함으로써 스케쥴링 오버헤드와 부하균등 사이의 tradeoff 관계를 최소화하였다. 본 논문의 구성은 2장에서 기존 스케쥴링 알고리즘에 대해 살펴보고 문제점들을 분석하며, 3장에서는 본 논문에서 제안된 스케쥴링 알고리즘을 제시하고, 4장에서는 기존 알고리즘들과 제안된 알고리즘에 대한 성능을 비교 평가한다. 마지막으로 5장에서는 결론 및 향후 연구 방향을 제안한다.

2. 관련 연구

병렬 스케쥴링 방법은 크게 전역 프로그램 정보인 매개변수를 토대로 컴파일 시간에 병렬 타스크들을 프로세서에게 균등하게 할당하는 정적 스케쥴링과 실행시간에 부하불균등을 고려하여 반복들을 프로세서에게 할당하는 동적 스케쥴링으로 구분된다. 정적 스케쥴링은 컴파일 시간에 반복들이 프로세서에게 할당되기 때문에 낮은 스케쥴링 오버헤드를 달성할 수 있으나, 각 타스크들의 실행시간이 서로 상이한 경우에는 부하불균등이 발생된다. 동적 스케쥴링은 각 타스크들의 예측할 수 없는 실행시간에 직면하여 부하균등을 달성하기 위해 시도되었으며, 어느 정도의 동기화 오버헤드를 수반한다. 따라서, 스케쥴링에서는 서로 독립된 각 반복들에 대해 최소의 동기화 오버헤드와 부하균등을 달성하는데 초점을 두며, 그에 따른 일반적인 루프 구조와 기존 스케쥴링 방법들을 살펴보면 다음과 같다.

2.1 루프 구조 분석

루프 구조는 모든 반복들간에 종속성이 없는 경우(DOALL)와 서로 다른 반복간에 종속성이 발생하는 경우(DOACR)가 존재한다[13,15]. DOALL 형태의 루프에서는 서로 다른 반복간에 종속 관계가 존재하지 않기 때문에 한 반복내의 모든 데이터를 프로세서간의 상호 작용없이 병렬 처리할 수 있다[14]. 즉, 부분적으로 프로세서간의 정보 교환이 필요하게 된다. DOACR 루프에서는 한 반복에서 가져온 데이터가 다른 반복에서 수정되거나 한 반복에서 생성된 결과가 나중에 다른 반복에서 사용되는 종속 관계가 존재하기 때문에 완전한 병렬 처리를 수행할 수 없다. 이러한 프로세서간의 상호 정보 교환을 동기화 기법이

라 하며, 프로그램의 올바른 수행을 위해서 반드시 유지되어야 한다. 다음은 DOALL과 DOACR 그리고 DOSERIAL문이 포함된 일반적인 루프 구조를 나타낸다.

(1) 1원 중첩 루프(One-Way Nested Loop)

같은 중첩 레벨에 하나의 루프만이 중첩된 루프 구조로서, 그림 1 (a)는 하나의 DOALL이 중첩 된 1원 중첩 루프를 나타낸 것이다. 그림1 (a)에서 중첩된 DOALL의 인덱스들은 합체될 수 있으며, 그림1 (b)는 합체된 루프를 나타낸 것이다. 합체된 루프에서 서로 다른 중첩된 레벨에 표현된 코드 분할(code segment)들의 실행 순서를 유지하기 위해서는 그림 1 (b)의 S_1은 합체된 루프의 조건에 따라 실행되어야 하며, 컴파일러는 S_1의 첫번째 문장 이전에 조건문을 삽입해야 한다.

```
  DOALL 1 J=1,N
        S1
     DOALL 2 K=1,N
          S2
  2  ENDOALL
1 ENDOALL
      (a)

  t=0
  DOALL  1 I=1,N^2
    IF (⌈I/N⌉ .NE. t) THEN
        S1
        t=⌈I/N⌉
    ENDIF
        S2
1 ENDOALL
      (b)
```

그림 1. 1원 중첩 루프와 합체

(2) 다원 중첩 루프(Multi-Way Nested Loop)

같은 중첩된 레벨에 둘 이상의 루프가 중첩된 루프 구조로서 그림 2 (a)는 두 개의 루프가 같은 중첩 레벨에 존재하는 2원 중첩 루프(2-way nested loop)를 나타낸 것이다. 그림 2 (a)에서 중첩된 두 개의 루프 인덱스들은 합체될 수 있으며, 그림 2 (b)는 합체된 루프를 나타낸 것이다. 합체된 루프에서 중첩된 두 문장의 실행 순서는 바뀌어야 하며, 따라서, A(J,*)는 B(J,*)의 어떤 요소가 산출되기 전에 계산되어야 한다.

```
   DOALL 1 J=1,N
      DOALL 2 K=1,N
         A(J,K)= . . .
   2  ENDOALL
      DOALL 3 L=1,N
         B(J,L)=. . .
   3  ENDOALL
 1 ENDOALL
      (a)

   DOALL 1 I=1,N^2
     A(⌈I/N⌉, I-N⌊(I-1)/N⌋) = . . .
     B(⌈I/N⌉, I-N⌊(I-1)/N⌋) = . . .
 1 ENDOALL
      (b)
```

그림 2. 2원 중첩 루프와 합체

(3) 혼합 루프(Hybrid Loop)

DOALL, DOACR 그리고 DOSERIAL 루프들이 혼합된 루프 구조로서, 그림 3 (a)는 DOSERIAL과 DOALL 루프가 중첩된 혼합 루프를 나타낸 것이다. 이런 경우, 루프 합체는 혼합 루프의 DOALL만이 적용되어질 수 있으며, 그림 3 (b)는 이에 상응하는 합체된 루프를 나타낸 것이다.

```
  DOALL 1 J=1, N
    DOSERIAL 2 K=1, N
       DOALL 3 L=1, N
          A(J,K,L) = . . .
    3  ENDOALL
  2 ENDSERIAL
1 ENDOALL
      (a)

  DOSERIAL 1 K=1, N
    DOALL 2 I=1, N^2
      A(⌈I/N^2⌉, K, I-N⌊(I-1)/N⌋) = . . .
  2 ENDOALL
1 ENDOSERIAL
      (b)
```

그림 3. 혼합 루프와 합체

2.2 기존 병렬 스케줄링 방법

(1) Self-Scheduling(SS)

SS[7,12]는 실행될 모든 반복들이 전역 메모리에 놓여지고, 프로세서가 idle하면, 중앙 작업 큐로부터 동기화 방법으로 루프 인덱스를 증가하며 하나의 반복을 선택하는 가장 단순한 스케줄링 방법이다. 따라서, 프로세서들이 각각 한 반복내에 끝나기 때문에 부하균등에 관해서는 최적의 동적 스케줄링 방법이다. 그러나 병렬 반복 수인 N dispatch operation이 발생하며, 전역 메모리의 상호 배타적인 접근이 빈번하여 오버헤드와 병목현상이 발생된다.

(2) Chunk Self-Scheduling(CSS)

CSS[1]는 전역 메모리의 상호 배타적인 접근의 빈번함을 줄이기 위해 idle 프로세서에게 반복의 그룹인 고정된 크기 즉, k의 chunk를 할당하여 실행함으로써 동기화 오버헤드를 줄이려고 하였다. 그러나 실행시간에 조차 알려지지 않은 각 반복의 분포 때문에 특히, 불규칙 루프에서 부하불균등이 발생되며, 또한 최적의 chunk를 구하기가 어렵다.

(3) Guided Self-Scheduling(GSS)

GSS[2]는 전역 메모리에 있는 실행될 반복을 idle 프로세서에게 할당할 때, 처음에는 큰 chunk를 할당함으로써 스케줄링 오버헤드를 감소시키고, 실행시간 동안 chunk의 크기를 계속 감소시킴으로써 부하균등을 달성하려고 하였다. GSS에서 각 프로세서들이 실행할 반복들에 대한 chunk의 크기는 다음과 같이 결정된다. 즉, 전역 메모리의 서로 독립된 반복들 $I_c = 1,2,3,4,....N$ 에 대해

$$R_0 = N, R_{i+1} = R_i - x_i$$

$$x_i = [\frac{R_i}{P}] \tag{1}$$

N : 총 반복 수, P : 프로세서 수, R_i : 실행되지 않은 반복 수, x_i : 할당될 반복 수

그러나 전역 메모리로부터 반복적으로 chunk를 계산하고 할당하기 위해, 프로세서들이 상호 배타적인 빈번한 메모리의 접근으로 많은 동기화 오버헤드가 발생된다. 또한 초기에 너무 큰 반복의 chunk가 프로세서에게 할당($\frac{N}{P}$)되기 때문에 반복들의 실행시간이 선형적으로 감소하거나, 처음 몇몇 반복들의 실행시간이 큰 경우 부하불균등이 발생되어 최적의 실행 시간내에 끝나지 않는다.

(4) Factoring

Factoring[8,9]은 반복들의 chunk가 P개의 프로세서로 스케줄링된 후, 전역 메모리에 충분한 반복들의 수 즉, PF_0를 남겨 부하균등을 달성하는 확률을 높이려고 하였다. Factoring에서는 전역 머모리에 있는 반복들에 대해 P개의 동일한 크기인 chunk들의 batch로 스케줄링된다. 따라서, 각 batch는 전역 메모리에 남이있는 반복들의 고정된 비율이 된다. Factoring에서 각 chunk는 다음과 같이 결정된다. 전역 메모리에 있는 서로 독립된 반복들 $I_c = 1,2,3,4,....N$ 에 대해

$$R_0 = N, R_{j+1} = R_j - PF_j$$

$$F_j = [\frac{R_j}{x_j P}] \text{ , } x_j = 2 \tag{2}$$

Factoring은 초기 chunk 크기를 줄이기 위해서 GSS의 $[\frac{N}{P}]$을 $[\frac{N}{2P}]$으로 하였으나, 초기 $[\frac{N}{2P}]$의 크기인 P개의 chunk중 $\frac{1}{2}$개 chunk내 반복들의 실행시간이 클때, 부하불균등이 발생되어 최적의 종료시간을 달성하지 못한다.

(5) Affinity Scheduling

Affinity Scheduling[3]은 반복에 대해 정적으로 $[\frac{N}{P}]$인 chunk를 각 프로세서에게 할당하고, 프로세서들은 각각 독립적으로 스케줄링을 수행함으로써 동기화에 대한 요구를 최소화 하려고하였다. 또한, 프로세서가 idle하면, 가장 많이 남아있는 프로세서를 찾은후, $[\frac{1}{P}]$의 반복을 옮겨 계속 실행함으로써 부하균등을 달성하려고 하였다. 그러나 Affinity Scheduling은 초기에 자료 국부성을 고려하지 않고 각 프로세서에게 정적으로 할당하였다. 따라서, 불규칙적인 경우에 각 chunk내 반복들의 분포가 상이하여 다른 프로세서에 접근하는 빈도가 많아지고, 접근된 프로세서로부터 작은 반복들($[\frac{1}{P}]$) 만을 취하여 실행하기 때문에 과도한 오버헤드가 발생되어 전체 성능에 나쁜 영향을 준다.

3. 새로운 병렬 루프 스케줄링 알고리즘

3.1. 이론 설명

본 논문에서는 서로 독립된 반복들에 대한 자료 국부성과 프로세서 동족성을 고려하여 반복들의 실행시간이 서로 상이한 경우에도 효과적으로 스케줄링을 할 수 있는 새로운 병렬 루프 스케줄링 알고리즘을 제안한다. 서로 다른 독립된 반복들은 컴파일 시간뿐만 아니라 실행시간에 조차 분포를 알 수 없기 때문에 부하불균등이 발생된다. 또한, 실행될 반복들의 chunk를 계산하여 프로세서에게 할당하기 위해 전역 메모리의 상호 배타적인 과도한 접근으로 스케줄링 오버헤드 및 메모리 경쟁이 유발된다. 이러한 부하불균등과 과도한 스케줄링 오버헤드는 결과적으로 스케줄링 성능에 나쁜 영향을 주게된다. 따라서, 부하균등을 달성하고 최소의 메모리 경쟁을 통한 스케줄링 오버헤드를 최소화함으로써 최적의 스케줄링을 달성하려는 데 목적이있다. 기존의 대부분 스케줄링 알고리즘들은 N개의 반복들 즉, [1,2,3,4,...., N-3, N-2, N-1, N]에 대해 서로 독립된 임의 변수로 취급하여 전역 메모리로부터 프로세서가 idle할 때마다 chunk 크기를 계산하여 프로세서에게 할당한다. 예를 들면, GSS[2]는 $[\frac{N}{P}]$, Factoring[8]은 $[\frac{N}{2P}]$ chunk를 전역 메모리가 공백일 때까지 반복적으로 계산하고 idle 프로세서에게 할당한다. 이는 빈번한 메모리의 상호 배타적인 접근과 반복된 chunk의 계산 및 할당으로 chunk의 수에 비례하는 스케줄링 오버헤드를 유발한다. Affinity Scheduling[3]은 $\frac{N}{P}$의 chunk를 정적으로 프로세서에게 할당하여 메모리의 상호 배타적인 접근 횟수를 줄여 스케줄링 오버헤드를 감소시키려고 하였다. 그러나 자료 국부성을 충분히 고려하지 않고 반복에 대한 chunk를 계산하고 할당하기 때문에 각 chunk내 반복들의 실행시간이 서로 상이하여 다른 프로세서의 메모리에 접근하는 횟수가 많아진다. 또한, 접근한 메모리로부터 적은 반복만을 취함으로써 전체적으로 스케줄링 오버헤드를 증가시켜 효율적인 스케줄링을 달성할 수 없다. 따라서, 최적의 스케줄링을 달성하기 위해서는 각 반복들에 대한 자료 국부성을 충분히 고려하여 분해함으로써 분해된 각 그룹간에 최소의 상이한 실행시간을 유지하도록 해야한다. 또한, 분해된 그룹들을 프로세서 동족성을 고려하여 프로세서들의 지역 메모리에 정적으로 할당하고, 각 프로세서는 자신의 그룹으로부터 연속된 반복들의 범위에 대해서 단일 루프 인덱스로 독립적인 실행을 함으로써 전체적으로 최소의 스케줄링 오버헤드와 부하균등을 달성할 수 있다.

3.2. 새로운 스케줄링 알고리즘

본 논문에서는 N개의 서로 독립된 반복 [1,2,3,4,..., N-3, N-2, N-1, N]에 대해, 자료 국부성을 고려하여 컴파일 시간에 정적으로 분해 작업을 수행하고, 프로세서에게 할당함으로써 동기화 오버헤드를 줄인다. 분해 작업의 기본 개념은 주어진 반복들을 제어 키에 의해 여러 개의 그룹으로 분해하고, 분해된 각 그룹들을 프로세서에 의해 독립적으로 스케줄링을 수행하게 하기 위한 것이다. 이때, 제어 키는 프로세서의 수가 된다. 예를 들면, 프로세서가 4인 다중프로세서 시스템인 경우를 고려 해보자. 첫째, 제어 키는 프로세서의 수가 되기 때문에 4가 된다. 따라서, 첫 번째 반복으로부터 제어 키 만큼 떨어진 반복들을 하나의 그룹으로 묶으면, 그림 4와 같이 4개의 그룹으로 분해된다.

[1,2,3,4,...., N-3, N-2, N-1, N]에 대해

$G_1 = [1, 1+CK, 1+2CK, \ldots, N-3]$
$\Rightarrow X[1], X[5], X[9], X[13]\ldots$
$G_2 = [2, 2+CK, 2+2CK, \ldots, N-2]$
$\Rightarrow X[2], X[6], X[10], X[14]\ldots$
$G_3 = [3, 3+CK, 3+2CK, \ldots, N-2]$
$\Rightarrow X[3], X[7], X[11], X[15]\ldots$
$G_4 = [4, 4+CK, 4+2CK, \ldots, N]$
$\Rightarrow X[4], X[8], X[12], X[16]\ldots$
CK: 제어 키(Control Key), X: 요소(Element)

그림 4. 제어키에 의한 정적 분해

따라서, G_i 번째 그룹의 i번째 요소를 X라 하면, 그 순서 위치는 다음과 같다.

$$X\,[(i-1)*CK+G_i] \quad (3)$$

분해된 각 그룹들은 N개의 서로 독립된 반복들에 대해서 자료 국부성을 고려했기 때문에 각 그룹간에 반복들의 실행시간에 따른 분포 차를 최소로 분해하

였다. 둘째, 분해된 그룹들은 각 프로세서의 지역 메모리로 할당되고, 프로세서들은 각각 자신의 지역 메모리로부터 독립적으로 반복들에 대해 스케줄링을 수행한다. 따라서, 프로세서들이 자신의 지역 메모리로부터 반복들을 독립적으로 수행하기 때문에 병목현상은 발생되지 않는다. 셋째, 만일 프로세서 P_i의 지역 메모리가 공백이 되어 idle하면, 프로세서 P_i는 나머지 P-1개의 프로세서 지역 메모리를 분석하고, 반복들이 가장 많이 남아 있는 프로세서의 지역 메모리의 반복들에 대해 다시 분해 작업을 수행한다. 이때, 분해될 반복들은 원래 수행하고 있던 프로세서와 idle 프로세서 P_i 즉, 2개의 프로세서에 의해 실행되므로 제어 키는 2가 된다. 따라서, 반복들은 두 개의 그룹으로 분해되고, 분해된 그룹은 각 프로세서에 의해 독립적으로 스케줄링이 반복 수행된다. 예를 들면, 프로세서 P_1이 idle하고, P_4의 지역 메모리가 가장 많은 반복들을 가지고 있다고 가정하면, 프로세서 P_4의 지역 메모리에 남아있는 반복들에 대한 분해 과정은 그림 5와 같다. 즉, P_4에 남아있는 반복들 $[24, 28, 32, 36,.., N-4, N]$에 대해

$$G_1 = [1, 1+CK, 1+2CK, ..., N-4]$$
$$\Rightarrow X[1], X[3], X[5], X[7]...$$
$$G_2 = [2, 2+CK, 2+2CK, ..., N]$$
$$\Rightarrow X[2], X[4], X[6], X[8]...$$

그림 5. 지역 메모리 반복들의 재분해

분해된 각각의 그룹에 대해서 프로세서 P_1은 G_1을 P_4은 G_2를 독립적으로 스케줄링을 수행한다. 이런 분해 작업은 인접된 반복들을 서로 다른 그룹으로 분리함으로써 각 그룹간의 실행시간에 따른 분포를 균일하게 유지하기 위한 것이다. 각 그룹간의 균일한 분포는 다른 프로세서의 지역 메모리 접근 횟수를 줄일 수 있으며, 따라서 동기화 오버헤드와 부하균등을 달성하여 최적의 스케줄링을 수행할 수 있다. 그 과정은 다음과 같다.

```
1. loop_initialization(N, CK)
   for i ← 1 to CK
     Gi ← ∅
      for j ← 1 to N  by CK //N 반복들을 CK개의
                그룹으로 분해//
         Gi ← Gi∪X(j)
      end
      CK ← CK-1
   end
2. for i ← 1 to CK //지역 메모리로 그룹 반복들을
          할당//
   Pi ← Gi
   end
3. loop
     for i ← 1 to CK//각 프로세서에 의해 독립적인
             스케줄링//
       if (Gi = empty)
         mlq= find_most_loaded_local_queue
         if (mlq=null) break;
           CK ← 2 //제어키에 2를 대입//
           for i ← 1 to CK-1
             Gi ← ∅
              for j ← 1 to N  by CK //가장 많이
                    남은 프로세서내 반복들의
                    재분해//
               Gi ← Gi ∪X(j)
              end
           end
         Pi ← Gi //idle 프로세서에게로 재분해된
                 반복 그룹 적재//
     end
   forever
```

그림 6. 새로운 병렬루프 스케줄링 알고리즘

4. 성능 평가

본 논문에서 제안된 스케줄링 알고리즘의 성능평가를 위해 4개의 벤치마크 프로그램을 선택하였다. 선택된 벤치마크 프로그램들은 성능평가의 공정성을 기하기 위해서 기존의 스케줄링 알고리즘들에서 이미 성능평가된 벤치마크 프로그램들을 다양하게 선택하였다. 즉, 그림 7은 Matrix Multiplication Program으로써 Factoring[8]에서 평가되었고, 그림 8과 그림 9는 Adjoint-convolution Program으로서 Factoring과 Affinity[3]에서 평가된 프로그램들이다. 그림 10은 루프 body부분의 조건문에 따라 각 반복들의 실행시간에 영향을 주는 벤치마크 프로그램으로 GSS[2]에서 평가된 프로그램이다. 선택된 벤치마크 프로그램들은 규칙 루프와 불규칙 루프를 모

두 포함시켰기 때문에 객관적인 성능 평가의 기준이 될 수 있다. 선택된 벤치마크 프로그램들의 성능 평가는 128개의 DEC Alpha chip으로 구성된 Cray T3E-900 CL128a-128인 MPP 머신에서 수행된다. 이 시스템은 자체 I/O Device를 가질 수 있도록 설계되어 독자적인 형태로 사용이 가능하며, 프로세서 성능은 450 MHz/PE이고, 프로세서당 128 MBytes의 지역 메모리를 가지고 있는 논리적으로 공유메모리 시스템이다.

본 논문에서는 성능 평가의 다양성과 공정성을 기하기 위해서 선택된 벤치마크 프로그램들을 2, 4, 8, 16개의 프로세서 상에서 실험을 수행하며, 이때 비교 평가될 각 알고리즘에 따른 벤치마크 프로그램들의 총 실행시간과 실행율을 비교 평가한다. 성능 평가를 위해 선택된 알고리즘들은 기존 스케쥴링 알고리즘들 중에서 비교적 성능이 우수한 것으로 평가된 GSS, Factoring, Affinity와 제안된 병렬루프 스케쥴링 알고리즘과의 성능을 비교 평가한다.

4.1. 벤치마크 프로그램

본 논문의 실험에서 사용된 4개의 벤치마크 프로그램은 그림 7부터 그림 10까지 나타나있다. 선택된 벤치마크 프로그램들은 각 반복들에 따른 다양한 실행 시간을 가진 벤치마크 프로그램들로써 성능평가의 공정성과 다양성을 제공할 수 있다.

```
    DOALL 10 I' = 1, N*N
      I = ⌈I'/N⌉
      J = I'-N*⌊(I'-1)/N⌋
      DOSERIAL 20 K= 1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
   20 ENDSERIAL
10 ENDOALL
```

그림 7. Matrix Multiplication Program

위 프로그램은 N*N개의 각 병렬 반복들이 내부의 순차 루프인 DOSERIAL 20 K= 1,N를 실행하며, 따라서 각 병렬 반복들은 $O(N)$의 동일한 실행시간을 가진다. 실험에서 위 프로그램 크기는 N=350일 때, 실행시간과 실행율에 대한 성능을 비교 평가한다.

```
DOALL 10 I=1, N*N
   DOSERIAL 20 K=I, N*N
      A(I)=A(I)+X*B(K)*C(K-I)
   20 ENDSERIAL
10 ENDOALL
```

그림 8. Adjoint-convolution Program1

```
  DOALL 10 I=N*N, 1,-1
    DOSERIAL 20 K=I, N*N
      A(I)=A(I)+X*B(K)*C(K-I)
 20 ENDSERIAL
10 ENDOALL
```

그림 9. Adjoint-convolution program2

위 프로그램들은 N*N개의 각 병렬 반복들이 내부의 순차 루프인 DOSERIAL 20 K=I, N*N을 실행한다. 따라서, 그림 8의 각 병렬 반복들은 $O(n^2- i)$에 비례하는 실행시간을 가지며, 그림 9의 각 병렬 반복들은 $O(n^2- i)$에 반비례하는 서로 다른 실행시간을 가진다. 실험에서 위 프로그램 크기는 N=350일 때, 실행시간과 실행율에 대한 성능을 비교 평가한다.

```
    DOALL 10 I=1, N
      IF [ C then {350} ]
      DOSERIAL 20 K=1, M
         {10}
   20 ENDSERIAL
10 ENDOALL
```

그림 10. 불규칙 루프

위 프로그램은 N개의 각 병렬 반복들이 C의 조건에 따라 350개의 직선 코드를 실행한 후, 내부의 순차 루프인 DOSERIAL 20 K=1, M을 실행한다. 따라서, 각각의 병렬 반복들은 C의 조건에 따라 서로 다른 실행시간을 가진다. 실험에서 위 프로그램 크기는 N=350일 때, 실행시간과 실행율에 대한 성능을 비교 평가한다. 또한, 각 병렬 반복들에 대해 앞의 몇몇 반복들의 분포가 클 때와 뒤의 몇몇 반복들의 분포가 클 때의 실행시간을 비교 평가하기 위해, C의 조건을 $I \le 90$와 $I \ge 310$인 두 가지 경우의 성능에 대한 평가를 수행한다.

4.2. 성능 분석 비교
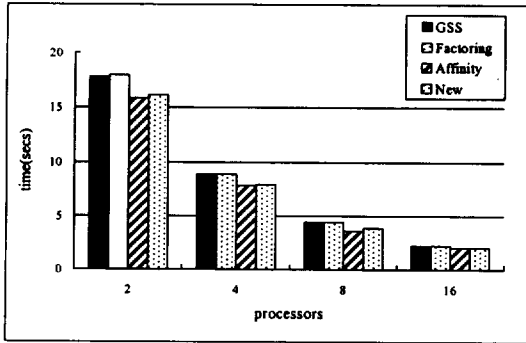
그림 11. Matrix Multiplication의 성능평가

표 1. Matrix multiplication의 실행시간

프로세서 수	GSS		Factoring		Affinity		New	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	17.79	89.09	17.96	88.25	15.85	100	16.13	98.98
4	8.89	79.66	8.89	87.96	7.82	100	7.90	98.98
8	4.45	81.12	4.44	81.30	3.61	100	3.87	93.28
16	2.23	90.58	2.22	90.99	2.02	100	2.04	99.01
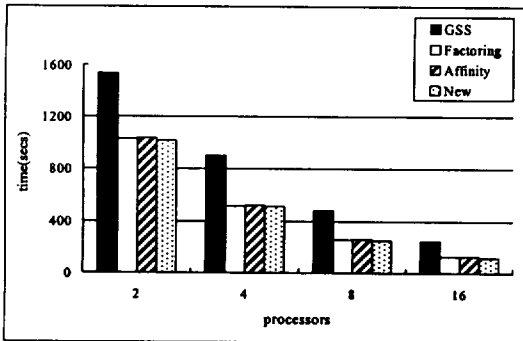
그림 12. Adjoint-convolution1의 성능평가

표 2. Adjoint-convolution1의 실행시간

프로세서 수	GSS		Factoring		Affinity		New	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	1538.77	65.88	1026.27	98.74	1033.56	98.04	1013.34	100
4	897.48	56.78	513.14	99.32	516.81	98.61	509.66	100
8	480.88	52.05	256.67	97.52	258.48	96.83	250.31	100
16	248.47	48.35	128.31	93.64	129.48	92.80	120.16	100
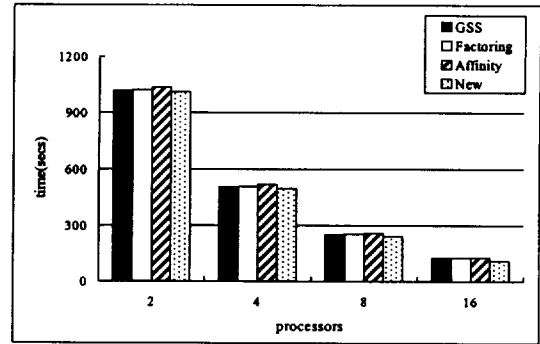
그림 13. Adjoint-convolution2의 성능평가

표 3. Adjoint-convolution2의 실행시간

프로세서 수	GSS		Factoring		Affinity		New	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	1017.42	99.20	1021.67	98.78	1033.55	97.65	1009.30	100
4	507.02	97.97	509.56	97.48	516.80	96.11	496.73	100
8	253.15	96.14	254.47	95.64	258.50	94.15	243.38	100
16	126.07	88.19	127.16	87.44	129.47	85.88	111.19	100
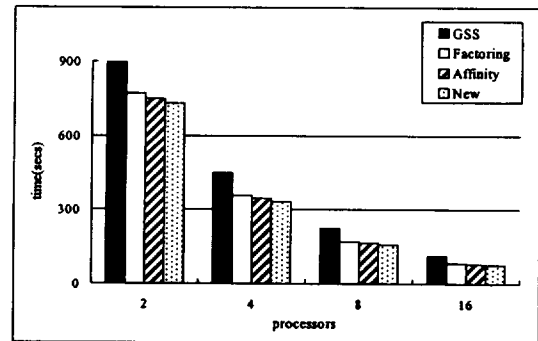
그림14. 불규칙 루프 I≤90의 성능평가

표 4. 불규칙 루프 I≤90일 때, 실행시간

프로세서 수	GSS		Factoring		Affinity		New	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	895.74	81.62	771.93	94.71	749.78	97.51	731.15	100
4	447.86	74.03	355.07	93.37	345.92	95.84	331.56	100
8	223.91	70.24	169.73	92.66	164.46	95.63	157.28	100
16	111.95	68.34	82.95	92.23	79.72	95.97	76.51	100
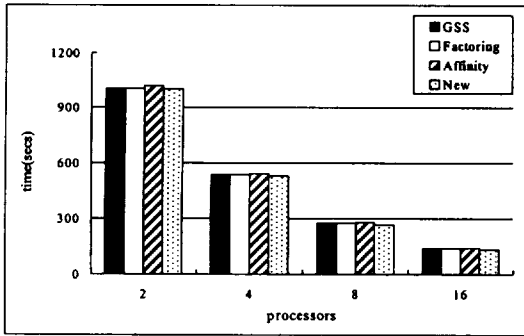
그림 15. 불규칙 루프 I≥310의 성능평가

표 5. 불규칙 루프 I≥310의 실행시간

프로세서 수	GSS		Factoring		Affinity		New	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	1002.82	99.80	1002.69	99.81	1019.73	98.14	1000.83	100
4	535.33	98.69	535.30	98.70	540.77	97.70	528.35	100
8	276.12	97.07	276.09	97.08	278.12	96.73	268.04	100
16	140.19	97.70	140.18	97.01	141.00	96.44	135.99	100

Matrix Multiplication의 실험 결과에서 Affinity와 제안한 알고리즘은 처음 정적으로 할당된 chunk의 분포가 동일하여 전역 메모의 접근이 발생되지 않아 좋은 수행결과를 보이고 있다. 그러나 GSS와 Factoring은 수행될 반복들의 chunk를 계산하기 위해 전역 메모리의 빈번한 접근으로 많은 오버헤드가 발생되어 P=2일 때, 각각 11%와 12%, P=4일 때, 20%와 12%, P=8일 때, 19%, 그리고 P=16일 때, 약 9%의 성능감소를 보이고 있다. Adjoint-convolution1의 실험 결과에서 GSS는 처음 할당된 chunk의 분포가 너무 커 부하불균등이 발생되며, Factoring은 각 반복들에 대해 부하균등은 달성되나 전역 메모리의 접근에 따른 오버헤드가 발생되고, Affinity는 처음 정적으로 할당된 chunk의 부하불균등으로 인해 다른 프로세서의 메모리에 접근하는 횟수가 많아 과도한 오버헤드가 발생된다. 따라서,제안된 알고리즘이 GSS와 Factoring, Affinity 보다 각각 11%, 3% 그리고 4%의 성능 향상을 보였다. Adjoint-convolution2의 실험결과에서 실험평가된 알고리즘 모두 비슷한 성능을 나타내고 있다. 그러나 GSS, Factoring, Affinity는 제안된 알고리즘 보다 과도한 스케쥴링 오버헤드가 발생된다. 따라서, 제안된 알고리즘은 약 4~5%의 성능 향상을 보였다. 그림 14인 불규칙 루프에서는 처음 각 프로세서에게 할당된 반복의 분포차이로 인해 GSS와 Factoring의 경우에는 부하불균등이 발생되며, Affinity의 경우에는 각 프로세서에 할당된 반복들이 부하불균등이 발생되어 그에 따른 각 프로세서의 종료시간을 유지하기 위해 과도한 전역 메모리에 접근하게 된다. 따라서, 제안된 알고리즘은 GSS, Factoring 그리고 Affinity에 비해 각각 약 26%, 7% 그리고 4%의 성능 향상을 보였다. 그림 15인 불규칙 루프에서는 실험 평가된 알고리즘 모두 비슷한 성능을 보였다. 그러나 제안된 알고리즘은 각 프로세서들에게 할당된 반복들 그룹의 분포 차가 비슷하여 다른 알고리즘보다 전역 메모리의 접근 횟수가 적게 발생되어 약 2~3%의 성능 향상을 보였다. 분석된 실험 결과에서 제안된 알고리즘은 다른 알고리즘에 비해 최소의 오버헤드로 부하균등을 달성할 수 있었으며, 각 프로세서는 많아야 1회의 전역 메모리 접근으로 모든 반복들의 수행을 완료함으로써 다른 알고리즘들보다 약 4~18%의 성능 향상을 보였다. 특히, 제안된 알고리즘은 각 반복들의 분포가 선형적으로 감소하거나 처음 몇몇 반복들의 분포가 클 때, GSS나 Factoring보다 좋은 성능 향상을 보였으며, 불규칙적인 루프에서 Affinity보다 더 좋은 성능향상을 보였다.

5. 결론 및 향후 과제

최적의 스케쥴링을 달성하기 위해서는 각 프로세서들이 반복들을 수행하기 위해 전역 메모리의 접근 횟수를 줄여 오버헤드를 최소화하고, 각 프로세서들이 반복들을 실행할 때, 각각 동일한 실행시간을 유지하여 부하균등을 달성해야 한다. 따라서, 이런 목적을 달성하기 위해서는 오버헤드를 최소화하는 정적 스케쥴링과 부하균등을 달성할 수 있는 동적 스케쥴링이 모두 고려된 스케쥴링 알고리즘이 되어야 한다. 본 논문에서는 동적 및 정적 스케쥴링을 모두 고려한 최적의 스케쥴링을 달성하기 위해 N개의 서로 독립된 반복들에 대한 자료 국부성을 고려함으로써 실행시간에 따른 분포가 거의 같은 반복들의 그룹으로 재구성하였다. 또한, 재구성된 그룹들은 프로세서 동족성을 고려하여 정적으로 각 프로세서의 지역 메모리에 할당하고, 할당된 각각의 그룹들을 독립적으로 스케쥴링을 수행함으로써 동기화 오버헤드를 줄

이고 부하불균등을 달성하는 새로운 병렬 루프 스케줄링 알고리즘을 제안하였다. 제안된 스케줄링 알고리즘은 실험 결과 각 프로세서마다 최대 1회의 전역 메모리 접근으로 부하균등을 유지할 수 있었으며, 동기화 오버헤드 및 병목현상을 최소화함으로써 전체적으로 GSS보다 P=2일 때, 13% P=4일 때, 19% P=8일 때, 21% P=16일 때, 22% 그리고 Factoring 보다 P=2일 때, 4% P=4일 때, 5% P=8일 때, 8% P=16일 때, 8%의 성능 향상을 보였으며, Affinity 보다는 P=2일 때, 2% P=4일 때, 3% P=8일 때, 5% P=16일 때, 7%의 향상된 스케줄링을 수행할 수 있었다. 또한, 제안된 스케줄링 알고리즘은 기존의 스케줄링 알고리즘과 비교 평가한 결과 최소 종료 시간 내에 모든 병렬 루프들을 수행함으로써 성능의 우수함을 입증하였다. 향후 과제로는 본 논문에서 제안된 스케줄링 알고리즘을 토대로 하여 각 서로 다른 프로세서의 성능 및 반복들의 전송 시간 등을 반영함으로써 공유 메모리 시스템에서뿐만 아니라 분산 메모리 시스템에서도 최적의 스케줄링을 달성할 수 있는 알고리즘에 대한 연구가 필요하다.

참 고 문 헌

[1] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processings," IEEE Transactions on Software Engineering, SE-11, October 1989.

[2] C. Polychronopoulos and D. Kuck, "Guided self-scheduling : A practical self-scheduling scheme for parallel supercomputers," IEEE Trans. Comput. vol. C-36, pp. 1425-1439, December. 1987.

[3] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor," IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 4 April 1994.

[4] L. M. Ni and C. F. E. Wu, "Designtrad-offs for processor scheduling in shared-memory multiprocessor systems," IEEE Trans. Software Eng., vol. 15, pp. 327-334, March. 1989. also in Proc. 1985 Int. Conf. Parallel Processing, pp. 63-70.

[5] M. S. Squillante, E. D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 2, February 1993.

[6] M. Weiss, Z. Fang, C. R. Morgan, and P. Belmont, "Effective dynamic scheduling and memory management on parallel processing systems," in proc. 1989 COMPSAC, pp. 122-129, September. 1989.

[7] P. Tang and P. C. Yew, "processor self-scheduling for multiple-nested parallel loops," in Proc. 1986 Int. Conf. parallel processing, pp. 528-535, August. 1986.

[8] S. F. Hummel, E. Schonberg, and L. E. Flynn. "Factoring: A practical and robust method for scheduling parallel loops," Comm. of the ACM35 (8) pp. 90-101, August. 1992.

[9] S. F. Hummal, I. Banicescu, C. Wang, and J. Wein, "Load balancing and data locality via Factoring: an Experimental Study," Proc. Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, pp. 85-89, Boleslaw K. Szymanski and Balaram Sinharoy(Editor), Kluwer Academic Publishers, Boston, May, 1995.

[10] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," Tech. Rep.MSU-CPS-ACS-27 Michigan State Univ., 1990.

[11] T. H. Tzen, L. M. Lionel, "Trapezoid Self-Scheduling : A Practical Scheduling for Parallel Compilers," IEEE Transaction on Parallel and Distributed System, vol. 4, no. 1, January 1993.

[12] Z. Fang, P. C Yew, P. Tang, and C. Q. Zhu, "Dynamic processor self-scheduling for general parallel nested loops," in Proc. 1987 Int, Conf. parallel processing, pp. 1-10, August. 1987.

[13] 박두순, 이광형, 황종선, 김병수, "병렬처리를 위

한 동기화 기법,"정보과학회지, 제13권 제7호, pp. 130-142, 7월 1995.
[14] 송월봉, 박두순, "최대 병렬성 추출을 위한 자료 종속성 제거 알고리즘,"정보과학회논문지(B), 제26권, 제1호, P139-149, 1999.
[15] 이광형, 황종선, 박두순, "중첩 루프의 병렬화를 위한 새로운 동기화 기법," 정보과학회 논문지(A), 제22권, 제11호, P1562-1570, 1995.

이 영 규

1989 대전산업대학교 전자계산학과 졸업(학사)
1992~1997 혜천대학 전자계산소 근무
1994 한남대학교 대학원 전자계산공학과 졸업(석사)
1998 순천향대학교 대학원 전산학과 박사과정 수료
1997~현재 극동정보대학 전산정보처리과 조교수
관심분야 : 병렬처리 컴파일러, 프로그래밍 언어론, 소프트웨어 공학 등임

박 두 순

1981 고려대학교 수학과 졸업(학사)
1983 충남대학교 대학원 계산통계학과 졸업(석사)
1988 고려대학교 대학원(전산학 전공) 졸업(박사)
1992~1993 미국 Univ. of Illinois at Urbana-Champaign CSRD 객원교수
1985~현재 순천향대학교 정보기술공학부 교수
관심분야 : 병렬처리 컴파일러, 계산이론, 프로그래밍 언어론 등임